Operator Overloading

The default and delete of constructors and member functions[1]

the `'=default;'` specifier to the end of a constructor declaration to declare that constructor as an explicitly defaulted constructor.

This makes the compiler generate the default implementations for explicitly defaulted constructors, which are more efficient than manually programmed constructors.

For example, whenever we declare a parameterized constructor, the compiler will not create a synthesized default constructor. In such a case, we can use the default specifier in order to create a default one.

```
class A {
public:

    // A user-defined parameterized constructor
    A(int x):val(x){}

    // ask the compiler to create the default
    // implementation of the constructor.
    A() = default;
    // A(A& obj) = default; // no need for this
private:
    int val;
};
```

If we do not define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy of its members. We need to define our own copy constructor only if an object has pointers.

Using the `'=default'` syntax uniformly for each of these special member functions makes code easier to read.

To disable the usage of a member function the `'=delete;'` specifier can be placed to the end of that function declaration.

Any member function disabled by `'=delete;'` is known as an explicitly deleted function.

Although not limited to them, but this is usually done to implicit functions.

[1] default and delete of constructors https://www.geeksforgeeks.org/explicitly-defaulted-deleted-functions-c-11/

```
class A {
public:
    A(int x): val(x){}

    // Delete the copy constructor
    A(const A&) = delete;

    // Delete the copy assignment operator
    A& operator=(const A&) = delete;
private:
    int val;
};
```

The class in the above example has no copy constructor nor copy assignment operator. They are deleted.

Input and Output Operators

Motivation: Streams for Built-in Types

The declaration of the overloading operators in the `class` called `ostream` similar to:

```
class ostream {
   public:
       ostream& operator<< (char);
       ostream& operator<< (int);
       ostream& operator<< (double);
       ...
};
```

The implementation:

```
ostream& operator<< (char) {
   // low-level functions to output characters
   return *this;
}
```

The return type is `ostream&`, not `ostream`. If we use the `ostream` as the return type, the `ostream` copy constructor will be called, but the `ostream` class **does not** have a copy constructor.

If you are the class writer for `ostream`, how can you design the class so it will not have a copy constructor? Make it private so that no one can use it, or delete it!

Declaration of the overloading operators in `class istream`

```
class istream {
   public:
       istream& operator>> (char&);
       istream& operator>> (int&);
       istream& operator>> (double&);
       ...
};
```

Q: again we return the reference of the first operand for the same reason as `cout`. But why do we pass a non-const reference? **That is,** we do:

```
istream& operator>> (char&);
```

but not

```
istream& operator>> (char);
istream& operator>> (const char&);
```

Because when we use `cin`:

```
char ans;
cin >> ans; // cin.operator>>(ans);
```

we want to modify `ans`.

Streams for User-defined Types: Non-Member Implementation

C++ Language Feature for Interpreting the Binary Operator

When compiler sees:

```
a*b
```

1. (**Member Implementation**) It can be interpreted in an object-oriented fashion so the first operand is the receiving object and the second operand is the passing argument of the message.

```
a.operator*(b)
```

2. (**Non-member Implementation**) It can also be interpreted as a global function combining two operands: both as passing arguments.

```
operator*(a, b)
```

Because we use the I/O operators (<< or >>) in the following way:

```
cin >> ans; // cin.operator>>(ans); // operator>>(cin, ans);
cout << ans; // cout.operator<<(ans); // operator<<(cout, ans);
```

The first operand is always the `istream` or `ostream` object.

For the user-defined types, if we want to define the I/O operators (<< and >>), the definitions should be **outside** of the user-defined class, as global functions.

```
#ifndef FRACTION_H
#define FRACTION_H

class Fraction {
  public:
    ...
  private:
    int numer;
    int denom;
};


inline
std::ostream& operator << (std::ostream& strm, const Fraction&
f){
  strm << f.numer << '/' << f.denom ;
  return strm;
}

...
#endif
```

We will talk about `inline` later. Notice that the non-member implementation of operator overloading usually need to read or write the nonpublic data members. As a consequence, classes often make these operator overloading global function **friend**s.

Summary

- If we wish to overload I/O operators for our own types, we must define them as nonmember functions.
- I/O operators usually read or write the non-public data members. As a consequence, classes often make the IO operators `friend`s.

Example: Fractions are numbers that can be written in the form *a*/*b*, where *a* and *b* are integers. *a* is known as the *numerator* and *b* the denominator. Implement a class `Fraction` and allow their objects to support the following functionalities:

```
#include <iostream>
#include "Fraction1.h"

using namespace std;

int main(){
    Fraction a(4,3);
    cout << "The fraction is: " << a << endl;
    return 0;
}
```

```
The fraction is: 4/3
```

```
#ifndef FRACTION_H
#define FRACTION_H
#include <iostream>

class Fraction {
friend std::ostream& operator << (std::ostream& out, const
Fraction& f);
public:
    Fraction(int n, int d): numer(n), denom(d){}
private:
    int numer;
    int denom;
};

inline
std::ostream& operator << (std::ostream& out, const Fraction& f){
    out << f.numer << '/' << f.denom;
    return out;
}

#endif // FRACTION_H
```

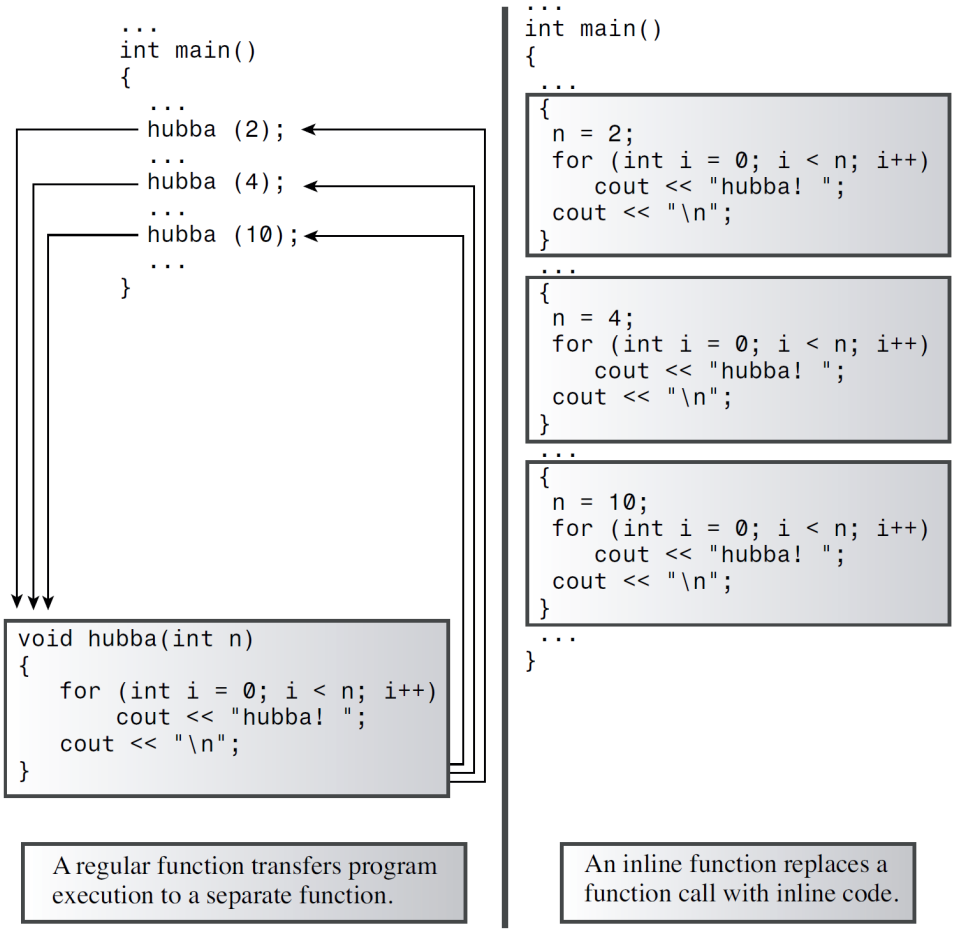
`inline` **Function**

Inline functions are a C++ enhancement designed to **speed up** programs. The primary distinction between normal functions and inline functions is not just in the keyword `inline`, but in how the C++ compiler incorporates them into a program.

Normal function calls involve having a program jump to another address (the function's address), execute the function, and then jump back when the function terminates.

C++ inline functions provide an alternative. In an inline function, the compiled code is **"in line" with the other code in the program**. That is, the compiler replaces the function call with the corresponding function code.

With inline, the program does not have to jump to another location to execute the code and then jump back. Inline functions thus run a little faster than regular functions, but they come with a memory penalty (see illustration below).



You should be selective about using inline functions. To use this feature, **you must take at least one of two actions**:

- Preface the function declaration with the keyword `inline`.
- Preface the function definition with the keyword `inline`. (this is a typical way)

The compiler does not have to honor your request to make a function inline. It might decide the function is too large or notice that it calls itself (recursion is not allowed or indeed possible for inline functions), or the feature might not be turned on in your compiler.

Normal non-inline functions can only be defined once. If you define your function twice in two separate source files, you will get a linker error.

In C++, each .cpp file is compiled independently first, then linked together. For `inline` functions, their **definitions** can be duplicated in each of the .cpps. The reason is because each of these .cpp files have to know the exact definition during compilation to perform the inline.

Duplicated definitions, however, must define exactly the same function. Not only should they be identical token by token, but the same name of a variable must refer to the very same object in each source file[2]. Remember that each .cpp could have its own static variable with the same name but independent copy? In other words, they are not the same object.

Example: A class `Fraction` with output operator overloading implemented as an `inline` function. We purposely put the implementation of the constructor in `Fraction.cpp` so `Fraction.h` will be included twice: one by the `main` client code and the other by `Fraction.cpp`.

Fraction.h

```
#ifndef FRACTION_H
#define FRACTION_H
#include <iostream>

class Fraction {
    friend std::ostream& operator << (std::ostream& out, const
Fraction& f);
public:
    Fraction(int n, int d);
private:
    int numer;
    int denom;
};

inline
std::ostream& operator << (std::ostream& out, const Fraction& f){
    out << f.numer << '/' << f.denom;
    return out;
}

#endif // FRACTION_H
```

[2] https://akrzemi1.wordpress.com/2014/07/14/inline-functions/

Fraction.cpp

```
#include "Fraction.h"

Fraction::Fraction(int n, int d) {
    numer = n;
    denom = d;
}
```

mainFraction.cpp

```
#include <iostream>
#include "Fraction.h"
using namespace std;

int main(){
    Fraction a(4, 3);
    cout << "The fraction is: " << a << endl;
    return 0;
}
```

```
The fraction is: 4/3
```

But if you remove `inline` from output operator overloading definition in `Fraction.h`, a linker error will rise indicating multiple definitions.

Arithmetic and Relational Operators

We define the arithmetic (`+ - * /`) and relational operators (`< > <= >= != ==`) as **non-member** functions. This allows the object of your class to be either the left- or right-hand operand.

```
y = x + 3;
y = 3 + x;
```

The arithmetic operators should not need to change the state of either operand, so the parameters are references to `const`.

An arithmetic operator usually generates a new value that is the result of a computation on its two operands. That value is **distinct** from either operand and is calculated in a local variable. The operation returns **a copy of this local as its result**.

Classes that define an arithmetic operator generally also define the corresponding **compound assignment operator**. When a class has both operators, it is usually more efficient to define the arithmetic operator to use compound assignment:

```
Sales_item operator+(const Sales_item &lhs, const Sales_item
&rhs){
    Sales_item sum = lhs; // copy data members from lhs into sum
    sum += rhs; // add rhs into sum
    return sum;
}
```

Compound Assignment Operator

Compound assignment operators are usually defined to be **member functions**. For consistency with the built-in compound assignment, these operators should **return a reference to their left-hand operand**.

```
Sales_item& Sales_item::operator+=(const Sales_item &rhs){
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```

Equality (==) and Less Than (<) Operators

Class used in associative containers **MUST** define the < operator. In general, a class should define the equality (==) and less than (<) operators since many STL algorithms assume these operators exist. For example, `sort` uses < and `find` uses ==.

Example: Write a `Fraction` class so we can insert its objects into a `set` for automatic sorting and output using `cout`. More info about `sets` can be found here[3].

```
#include <iostream>
#include <set>
#include "Fraction.h"

using namespace std;

int main(){
    set<Fraction> coll =
       {Fraction(2,3),Fraction(1,2),Fraction(3,4),Fraction(1,5),
        Fraction(4,3)};
    cout << "Fractions in a sorting order are: ";
    for (auto e : coll)
        cout << e << " ";
    cout << endl;
    return 0;
}
```

[3] URL: http://www.cplusplus.com/reference/set/set/

```
Fractions in a sorting order are: 1/5 1/2 2/3 3/4 4/3
```

A:

```
#ifndef FRACTION_H
#define FRACTION_H
#include <iostream>

class Fraction {
friend std::ostream& operator << (std::ostream& out, const
Fraction& f);
friend bool operator < (const Fraction& f1, const Fraction& f2);
public:
    Fraction(int n, int d):numer(n),denom(d){}
private:
    int numer;
    int denom;
};

inline
std::ostream& operator << (std::ostream& out, const Fraction& f){
    out << f.numer << '/' << f.denom;
    return out;
}

inline
bool operator < (const Fraction& f1, const Fraction& f2) {
    if (f1.numer*f2.denom < f2.numer*f1.denom) return true;
    else return false;
    // return (f1.numer*f2.denom < f2.numer*f1.denom) ? true :
false;
}

#endif // FRACTION_H
```

Subscript Operator

Classes that represent containers from which elements can be retrieved by position often define the subscript operator, `operator[]`. The subscript operator **MUST** be a member function.

A class that defines subscript needs to define **two** versions:

- one is a non-`const` member function that returns a reference (so that the subscript can appear on either side of the assignment operator), and
- the other is a `const` member function[4] that returns a `const` reference (so that when

[4] Content from https://www.geeksforgeeks.org/const-member-functions-c/
A function becomes `const` when `const` keyword is used in function's declaration. The idea of `const` functions is not allow them to modify the object on which they are called.
When a function is declared as `const`, it can be called on any type of object. Non-`const` functions can only

we subscript a `const` object, it cannot appear on the **left-hand-side** of the assignment operator).

```
#include <iostream>
#include <vector>
using namespace std;

class FooVec {
public:
    int &operator[] (const size_t index) { return data[index] }
    const int &operator[] (const size_t index) const {
        return data[index];
    }
    // other interface members
    FooVec(int i = 0); // allocate space in vector<int> data
private:
    std::vector<int> data;
    // other member data and private utility functions
};

int main(){
    FooVec f1(2);
    f1[0] = 3;
    f1[1] = f1[0];

    const FooVec f2(1);
    f1[1] = f2[0]; // ok
    f2[0] = f1[1]; // not ok!
}
```

be called by non-`const` objects.